PROVING CORRECTNESS OF COMPILER OPTIMIZATIONS BY TEMPORAL LOGIC

David Lacey (Oxford University)
Neil D. Jones (University of Copenhagen)
Eric Van Wyk (Oxford University)
Carl C. Frederiksen (University of Copenhagen)

July 5, 2010

- Conference paper: POPL 2002
- Journal paper:

  HOSC (Higher-Order and Symbolic Computation) 17(3), pp. 173-206. 2004.

CLASSICAL COMPILER OPTIMIZATIONS

Most work by rewriting intermediate code. e.g., by the "Dragon book":

1. dead code elimination
2. constant folding
3. code motion and many more...

Widely used, a practical success. How do they work ? Pretty well.

What are their techniques?

- Data-flow lattices, usually greatest fixpoints
- "`gen`" and "`kill`" sets
- "definitions," "uses,", "transparency"
- forwards and backwards analyses
- worklist algorithms, etc etc etc

Alas: much arcane technology, with no semantic explanations!

THE QUESTION OF CORRECTNESS

Schmidt, Muchnick, Jones, others in the 1970's:

"Are these compiling analyses and transformations correct?"

Consequence: much thought and research.

1. **What is "correct" ? . . . must be "preserves program semantics".**
2. **What is "program semantics" ? Hmm. . .**
   We learn denotational, then operational semantics.
3. **Are the "data-flow based program analyses" correct ? And what does this mean, anyhow ? Hmm. . . we learn/invent abstract interpretation.**
4. **Are "compiling transformations" correct? . . . What was the question again?**

Alas, too many complex pieces to put together.

Year 2000 (about): Enter Oege De Moor, with a nifty transformational way to describe compiler optimizations.

REWRITE RULES FOR COMPILER OPTIMIZATIONS

$$\boxed{\mathcal{I} \Rightarrow \mathcal{I}' \text{ if } \phi}$$

- $\mathcal{I}, \mathcal{I}'$ are intermediate language instructions
- ϕ is a data+control flow property, expressed in a temporal logic.

Reading:

- If program π has instruction $\mathcal{I}$ at a control point p, and
- flow condition ϕ holds at p, then replace $stmt$ by $\mathcal{I}'$.

Program

$$\pi = \boxed{\ldots p : \mathcal{I} \ \ldots}$$

is rewritten into:

$$\pi' = \boxed{\ldots p : \mathcal{I}' \ldots}$$

A program-centric transformation description:

- p is "here" in the program. Enabling property.
- ϕ says "what I can/must see from here," looks forwards or backwards in the program's flow chart.

THE GOAL OF THIS WORK

A way to prove that such a transformation is correct.

Correctness:

IF program π' = the result of rewriting program π by applying this rule:

$$\boxed{\mathcal{I} \Rightarrow \mathcal{I}' \text{ if } \phi}$$

THEN π and π' have exactly the same semantics:

$$\boxed{[\![\pi]\!] = [\![\pi']\!]}$$

For example, they compute the same input-output function, and have the same termination properties.

MOTIVATION

- The use of either
  - rewrite rules to capture transformation or
  - temporal logic to prove correctness of compiler optimizations

  is not new (Boyle, Steffen, etc.). What's new:
- Temporal logic plays a central role in our framework, allowing much easier correctness proofs of classical optimizing transformations.
- The transformations are well-suited to automation. They can be (and have been) implemented in optimizing compilers. (De Moor et al)
- Specification and implementation come closer, increasing confidence in the compiler.
- Lerner, Millstein, Chambers have further automated our approach including proofs, using their D.S.L. Cobalt to express (weaker) rewrite conditions.

  (They also found many bugs in "obviously correct" rules – applicability conditions that needed strengthening to preserve semantics!)

MEANS

- Temporal logic can describe properties of state transition systems.
- The run-time state space of a program is a state transition system.
- Temporal side conditions (e.g., to apply a rewrite) relate to an anonymous "current state."
- Atomic propositions tell information about the current state.
- Future: Next-state quantifiers:
  AX means "for every next state,"
  EX means "for some next state" (branching-time, so we used CTL)
- Past: Previous-state quantifiers:
  $\overleftarrow{AX}$ means "for every previous state,"
  $\overleftarrow{EX}$ means "for some previous state"
- Path quantifiers refer to some or all computational futures or pasts.

DEAD CODE ELIMINATION

Goal: remove assignment statement that assigns a value that's never used:

$$\boxed{\mathsf{x} := \mathsf{e} \implies \texttt{skip}}$$

Side condition on the rewrite: The value assigned is never referenced again after the current (anonymous) program point.

The rewrite rule, with forward path quantifier A in its side condition ϕ:

$$\boxed{\mathsf{x} := \mathsf{e} \implies \texttt{skip if } A(\neg use(\mathsf{x}))}$$

A more liberal condition:

$$\boxed{\mathsf{x} := \mathsf{e} \implies \texttt{skip if } AX\ A(\neg use(\mathsf{x}))}$$

Use AX operator to start at the next point, since there's no harm if x is used in e.

A still more liberal condition with "weak until":

$$\boxed{\mathsf{x} := \mathsf{e} \implies \text{skip if } AX\ A(\ \neg use(\mathsf{x})\ W\ [def(\mathsf{x}) \wedge \neg use(\mathsf{x})]\ )}$$

PROGRAM SYNTAX

A program π has form:

$\pi =$ `read input;` $\mathcal{I}_1; \mathcal{I}_2;$ $\ldots$ $\mathcal{I}_{m-1};$ `write output;`

where

- $\mathcal{I}_1, \ldots \mathcal{I}_{m-1}$ **are instructions**
- **program labels:** $labels(\pi) = \{1, \ldots, m\}$
- **1 labels the first statement in π.**
- $m = exit(\pi)$ **labels the concluding** `write y;`

Grammar:

$$\mathcal{I} \in stmt \ ::= \ x := e \mid \texttt{skip} \mid \texttt{if } x \texttt{ goto } p_1 \texttt{ else } p_2$$

$$e \in \exp \ ::= \ c \mid x \mid \texttt{op}(e_1, \ldots, e_n)$$

$$\begin{array}{lll} x & \in & Variable \\ p_1, p_2 & \in & \{1, \ldots, labels(\pi)\}. \end{array}$$

OPERATIONAL SEMANTICS OF A PROGRAM

The semantic transition relation

$$\boxed{\rightarrow \subseteq PgmState \times PgmState}$$

for a program $\pi \in pgm$ is defined by:

1. **If $\mathcal{I}_t = \texttt{skip}$ then** $\boxed{(t, \sigma) \rightarrow (t+1, \sigma)}$
2. **If $\mathcal{I}_t = (\texttt{x} := e)$ then** $\boxed{(t, \sigma) \rightarrow (t+1, \sigma[\texttt{x} \mapsto [\![e]\!]\sigma])}$
3. **If $\mathcal{I}_t = (\texttt{if x goto } p_1 \texttt{ else } p_2)$ and $\sigma(\texttt{x}) = \texttt{true}$ then** $\boxed{(p, \sigma) \rightarrow (p_1, \sigma)}$
4. **If $\mathcal{I}_t = (\texttt{if x goto } p_1 \texttt{ else } p_2)$ and $\sigma(\texttt{x}) \neq \texttt{true}$ then** $\boxed{(p, \sigma) \rightarrow (p_2, \sigma)}$
5. $\boxed{(exit(\pi), \sigma) \rightarrow (exit(\pi), \sigma)}$

The initial state is:

$$initial_\pi(v) = (1, [\texttt{input} \mapsto v, y_1 \mapsto true, \ldots, y_k \mapsto true])$$

where $vars(\pi) \setminus \{\texttt{input}\} = \{y_1, \ldots, y_k\}$.

COMPUTATIONS

A computation prefix is a finite or infinite sequence $C \in PgmState^{*\omega}$:

$$C = \boxed{\pi, v \vdash (p_0, \sigma_0) \longrightarrow (p_1, \sigma_1) \longrightarrow \ldots}$$

such that

- $(p_0, \sigma_0) = initial_\pi(v)$ and
- $(p_i, \sigma_i) \longrightarrow (p_{i+1}, \sigma_{i+1})$ for all $i \geq 0$

The semantic function $[\![\pi]\!] : Value \longrightarrow Value$ is partial:

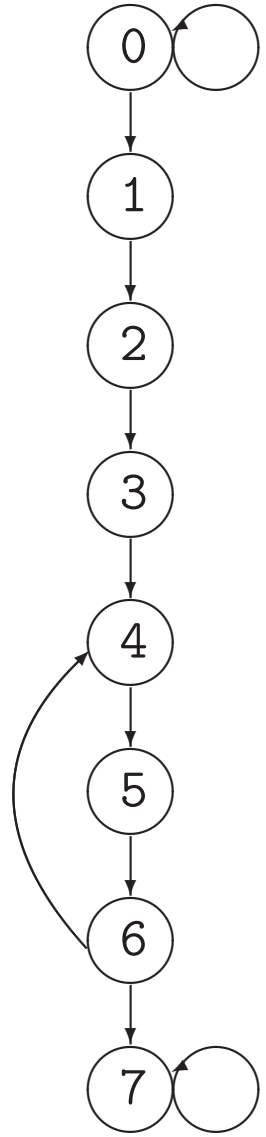
$[\![\pi]\!]v = \sigma_k(\mathrm{y})$ if there is a computation prefix

$$C = \boxed{\pi, v \vdash (p_0, \sigma_0) \longrightarrow \ldots \longrightarrow (p_k, \sigma_k)}$$

with $p_k = exit(\pi)$.

EXAMPLE PROGRAM AND CONTROL FLOW MODEL

```
     read x;

1:   five := 5;

2:   y := 0;

3:   c := five;

4:   y := y + c * x;

5:   x := x-1;

6:   if x then 4 else 7;

7:   write y;
```



(Loops at entry and exit for compatibility with temporal logic.)

CTL WITH FREE VARIABLES

In this talk: use an extended definition of CTL, CTL-FV to express conditions on the control flow. The extension is two-fold:

1. Reverse modality: Admits expressing backwards analyses, e.g., available expressions or constant propagation.

2. Propositions may have free variables, that are bound during model-checking to bits of the program being analyzed.

   - Implicit $\exists$ allows specification of rule schemata.
   - Explicit $\exists, \forall$ can be added (APLAS 2007).
   - OK as long as variable range is of bounded variation, e.g., bound to bits of or pointers to the program being analysed.

CTL WITH FREE VARIABLES

A CTL-FV model is a triple

$$\mathcal{M} = (S, \rightarrow, V)$$

where

1. S is a set of states
2. $\rightarrow \subseteq S \times S$ is a relation between states (forward and backward total).
3. The function $V : S \rightarrow 2^{AP}$ is called the valuation.

   It maps states to atomic propositions that hold in the given state.

A forward path has form

$$n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow \ldots$$

A backward path has form

$$\ldots \rightarrow n_2 \rightarrow n_1 \rightarrow n_0$$

CONTROL FLOW MODEL FOR EXAMPLE PROGRAM

```
 read x;
1: five := 5;
2: y := 0;
3: c := five;
4: y := y + c * i;
5: x := x-1;
6: if x then 4 else 7;
7: write y;
```

Compiler-writers' jargon:

- X is "defined" if program contains `X := ...`
- X is "used" if program contains `_:= ...X...`

Transitions and atomic propositions:

S	$=$	$\{1, 2, 3, 4, 5, 6, 7\}$
$\rightarrow$	$=$	$\{1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 4, 4 \rightarrow 5, 5 \rightarrow 6, 6 \rightarrow 7, 6 \rightarrow 4, 7 \rightarrow 7\}$
$V(1)$	$\supseteq$	$\{\mathrm{node}(1), \mathrm{stmt}(\texttt{five := 5}), \mathrm{def}(\texttt{five}), \mathrm{conlit}(5)\}$
$V(2)$	$\supseteq$	$\{\mathrm{node}(2), \mathrm{stmt}(\texttt{y := 0}), \mathrm{def}(\texttt{y}), \mathrm{conlit}(0)\}$
$V(3)$	$\supseteq$	$\{\mathrm{node}(3), \mathrm{stmt}(\texttt{c := five}), \mathrm{def}(\texttt{c}), \mathrm{use}(\texttt{five})\}$
$V(4)$	$\supseteq$	$\{\mathrm{node}(4), \mathrm{stmt}(\texttt{y := y + c * i}), \mathrm{def}(\texttt{y}), \mathrm{use}(\texttt{y}), \mathrm{use}(\texttt{c}), \mathrm{use}(\texttt{i})\}$
$V(5)$	$\supseteq$	$\{\mathrm{node}(5), \mathrm{stmt}(\texttt{x := x - 1}), \mathrm{def}(\texttt{x}), \mathrm{use}(\texttt{x}), \mathrm{conlit}(1)\}$
$V(6)$	$\supseteq$	$\{\mathrm{node}(6), \mathrm{stmt}(\texttt{if x then 4 else 7}), \mathrm{use}(\texttt{x})\}$
$V(7)$	$\supseteq$	$\{\mathrm{node}(7), \mathrm{stmt}(\texttt{write y}), \mathrm{use}(\texttt{y})\}$

OPERATIONAL INTERPRETATION

A model checker will

- **not only find which nodes in a model satisfy a state formula, but also**
- **find instantiations of CTL variables that satisfy the formula.**

Technically: extend satisfaction relation $n \models \phi$ **to** $n \models_\theta \phi$. **Substitution** θ **binds the free variables of** ϕ**, so**

$$\boxed{n \models_\theta \phi \text{ holds for any } \theta \text{ such that } n \models \theta(\phi)}$$

The job of the model checker: given ϕ**, to return the set of all** n **and** θ **such that** $n \models_\theta \phi$.

For the example program and formula

$$\boxed{\phi = def(x) \wedge use(x)}$$

the model checker returns the following set:

$$\{\theta_1, \theta_2\} = \{[n \mapsto 4, x \mapsto \mathrm{y}], [n \mapsto 5, x \mapsto \mathrm{x}]\}$$

CTL-FV SYNTAX

CTL-FV formulas are either state or path formulas generated by the grammar with productions:

$$\phi ::= true \mid ap(\mathsf{x}_1, \ldots, \mathsf{x}_n) \mid \neg\phi \mid \phi_1 \wedge \phi_2$$

$$\begin{array}{lll} \mid & A\psi & \text{for all forward paths} \\ \mid & E\psi & \text{for some forward path} \\ \mid & \overleftarrow{A}\psi & \text{for all backward paths} \\ \mid & \overleftarrow{E}\psi & \text{for some backward paths} \end{array}$$

$$\begin{array}{lll} \psi ::= & X\phi & \\ \mid & \phi_1 \, U \phi_2 & \phi_1 \text{ holds until W} \\ \mid & \phi_1 W \, \phi_2 & \phi_1 \text{ holds weakly until W} \end{array}$$

$$\mathsf{x} \in \mathit{CTL\text{-}FV\text{-}variable}$$

SATISFACTION IN CTL-FV

Let $\mathcal{M}$ be a model. Then:

State Formulas:

$\mathcal{M}, n \models_\theta true$ **iff** $true$

$\mathcal{M}, n \models_\theta ap(x_1, \ldots, x_n)$ **iff** $ap(\theta x_1, \ldots, \theta x_n) \in V(n)$

$\mathcal{M}, n \models_\theta \neg\phi$ **iff** **not** $\mathcal{M}, n \models_\theta \phi$

$\mathcal{M}, n \models_\theta \phi_1 \wedge \phi_2$ **iff** $\mathcal{M}, n \models_\theta \phi_1$ **and** $\mathcal{M}, n \models_\theta \phi_2$

$\mathcal{M}, n \models_\theta E\ \psi$ **iff** $\exists path\ (n = n_0 \rightarrow n_1 \rightarrow \ldots) : \mathcal{M}, (n_i)_{i\geq 0} \models_\theta \psi$

$\mathcal{M}, n \models_\theta A\ \psi$ **iff** $\forall path\ (\ldots \rightarrow n_1 \rightarrow n_0 = n) : \mathcal{M}, (n_i)_{i\geq 0} \models_\theta \psi$

$\mathcal{M}, n \models_\theta \overleftarrow{E}\ \psi$ **iff** $\exists path\ (n = n_0 \rightarrow n_1 \rightarrow \ldots) : \mathcal{M}, (n_i)_{i\geq 0} \models_\theta \psi$

$\mathcal{M}, n \models_\theta \overleftarrow{A}\ \psi$ **iff** $\forall path\ (\ldots \rightarrow n_1 \rightarrow n_0 = n) : \mathcal{M}, (n_i)_{i\geq 0} \models_\theta \psi$

Path Formulas:

$\mathcal{M}, (n_i)_{i\geq 0} \models_\theta X(\phi)$ **iff** $\mathcal{M}, \models_\theta \phi$

$\mathcal{M}, (n_i)_{i\geq 0} \models_\theta \phi_1\ U\ \phi_2$ **iff** $\exists i \geq 0 : \mathcal{M}, n_i \models_\theta \phi_2 \wedge \forall 0 \leq j < i : \mathcal{M}, n_j \models_\theta \phi_1$

$\mathcal{M}, (n_i)_{i\geq 0} \models_\theta \phi_1\ W\ \phi_2$ **iff** $\exists i \geq 0 : \mathcal{M}, n_i \models_\theta \phi_2 \wedge$
$\forall 0 \leq j < i : \mathcal{M}, n_j \models_\theta \phi_1 \vee \quad (\forall k \geq 0 : n_k \models_\theta \phi_1 \wedge n_{k+1}$ **exists**$)$

REWRITING

A rewrite rule has form $\mathcal{I} \Rightarrow \mathcal{I}'$ if ϕ, where $\mathcal{I}$ and $\mathcal{I}'$ are instructions built from program and CTL variables, and ϕ is a CTL-FV formula.

Define

$$Apply(\pi, \pi', n, \mathcal{I} \Rightarrow \mathcal{I}' \text{ if } \phi)$$

to be true if and only if for some substitution θ, the following hold:

1. $\mathcal{M}, n \models_\theta stmt(\mathcal{I}) \wedge \phi$

2. $\pi =$ `read input; 1: ... n: θ(I); ... write output;`

3. $\pi' =$ `read input; 1: ... n: θ(I'); ... write output;`

where $1 \leq n \leq m + 1$ and the $\ldots$'s are unchanged.

DEAD CODE ELIMINATION

Goal: **remove an assignment statement that assigns a value that is never used:**

$$\mathsf{x{:=}e} \quad \Longrightarrow \quad \texttt{skip}$$

Side condition on the rewrite: **the value assigned is never referenced again before it is (re-)defined.**

The rewrite rule with its side condition is written

$\mathsf{x{:=}e} \quad \Longrightarrow \quad \texttt{skip}$
if
$AX\ A(\ \neg use(\mathsf{x})\, W\, def(\mathsf{x}) \wedge \neg use(\mathsf{x})\ )$

CONSTANT FOLDING

Constant folding transformation replaces a variable reference with a constant value:

$$\mathbf{x{:=}y} \quad \Longrightarrow \quad \mathbf{x{:=}c}.$$

To check: whether all assignments to y assign it the same constant value.

To check this condition we use the past temporal operators as follows:

$$\mathbf{x{:=}y} \quad \Longrightarrow \quad \mathbf{x{:=}c}$$

if

$$\overleftarrow{A}\,(\ \neg \mathtt{def}(\mathbf{y}) \wedge \neg \mathtt{stmt}(\mathbf{read}\ \_)$$
$$U\ \mathit{stmt}(\mathbf{y{:=}c}) \wedge \mathit{conlit}(\mathbf{c}))$$

Point: to ensure that all paths

- from program entry "read "
- to instruction x:=y

contain a definition y:=c. By definition of $\models_\theta$ all the c's must be the same.

REWRITING AT MORE THAN ONE POINT

We may specify several rewrites and side conditions at once.

Example: To transform two nodes the form of the rewrite would be:

$$
\begin{array}{l}
\quad n : \mathcal{I}_1 \Longrightarrow \mathcal{I}'_1 \\
\quad m : \mathcal{I}_2 \Longrightarrow \mathcal{I}'_2 \\
\text{if} \\
\quad n \models \phi_1 \\
\quad m \models \phi_2
\end{array}
$$

Operational interpretation: find a substitution θ that satisfies

$$n \models_\theta stmt(\mathcal{I}_1) \wedge \phi_1 \text{ and } m \models_\theta stmt(\mathcal{I}_2) \wedge \phi_2$$

and use θ to alter the program in places n and m.

CODE MOTION/LOOP INVARIANT HOISTING

A double rewrite:

p : `skip` $\Longrightarrow$ **x:=e**

q : **x:=e** $\Longrightarrow$ `skip`

if

$p \models A(\neg use(\mathbf{x})\ W\ node(q))$

$q \models \neg use(\mathbf{x}) \wedge$

$\overleftarrow{A}((\neg def(\mathbf{x}) \vee node(q)) \wedge$

$trans(\mathbf{e}) \wedge \neg \mathtt{stmt}(\textbf{read x})\ W\ node(p))$

Example: **lift** `x:=a+b;` **from label** 3 **to label** 1.

```
1:  skip;
2:  if...then 3 else 6;
3:  x := a + b;        (inside a loop)
4:  y := y − 1;
5:  if y then 3 else 6;
6:  x := 0;
```

Optimized code: **lift** `x:=a+b;` **from label** $q = 3$ **to label** $p = 1$.

A METHOD FOR SEMANTIC EQUIVALENCE

To show: $Apply(\pi, \pi', p, \mathcal{I} \Rightarrow \mathcal{I}' \text{ if } \phi)$ **implies** $[\![\pi]\!] = [\![\pi']\!]$, **i.e..** $\forall$ **input** v

$initial_\pi(v) \rightarrow^* (exit, \sigma)$	**for program** π **if and only if**
$initial_{\pi'}(v) \rightarrow^* (exit, \sigma')$	**for program** π'**, and** $\sigma(\texttt{output}) = \sigma'(\texttt{output})$

The problem: how to link the
- **temporal** property ϕ ("futures" and "pasts") to
- the transformation $I \Longrightarrow I'$.

Solution: enrich the semantics and its transition system to **computation prefixes**:

$$C = \boxed{\pi, v \vdash s_0 \rightarrow \ldots \rightarrow s_t}$$

Prefix transition system $\mathcal{T}_{pfx}(\pi, v)$:
Define

$$C \rightarrow C_1 \in \mathcal{T}_{pfx}(\pi, v)$$

if and only if

$$C_1 = \boxed{\pi, v \vdash s_0 \rightarrow \ldots \rightarrow s_t \rightarrow s_{t+1}}$$

PROOF METHOD

Programs π and π' are semantically equivalent: $[\![\pi]\!] = [\![\pi']\!]$ if there exists a bisimulation relation $\mathcal{R}$ on computation prefixes, such that for all v:

1. **(Base Case) $\mathcal{R}$ holds between the initial computation prefixes, i.e.,**

$$\boxed{((\pi, v \vdash \mathit{initial}_{\pi}(v)) \;\; \mathcal{R} \;\; (\pi', v \vdash \mathit{initial}_{\pi'}(v)))}$$

2. **(Step) If $C_1 \;\mathcal{R}\; C_1'$ and $C_1 \longrightarrow C_2$ and $C_1' \longrightarrow C_2'$ then $C_2 \;\; \mathcal{R} \;\; C_2'$.**
3. **(Equivalence) If**

$$\boxed{\begin{array}{l} C \;\mathcal{R}\; C' \text{ and} \\ C = \pi, v \vdash s_0 \longrightarrow s_1 \ldots \longrightarrow (p_t, \sigma) \text{ and} \\ C' = \pi', v \vdash s_0' \longrightarrow s_1' \ldots \longrightarrow (p_{t'}, \sigma') \end{array}}$$

then

$$\boxed{\begin{array}{l} (i) \;\; p_t = \mathit{exit}(\pi) \iff p_{t'}' = \mathit{exit}(\pi') \text{ and} \\ (ii) \;\; p_t = \mathit{exit}(\pi) \wedge p_{t'}' = \mathit{exit}(\pi') \\ \qquad \Rightarrow \sigma_t(\mathtt{output}) = \sigma_{t'}'(\mathtt{output}) \end{array}}$$

DEAD CODE ELIMINATION

$$\phi \equiv AX\ A(\ \neg use(\mathsf{x}) W\, def(\mathsf{x}) \wedge \neg use(\mathsf{x})\ )$$

Consider $C \in \mathcal{T}_{pfx}(\pi, v)$ and $C' \in \mathcal{T}_{pfx}(\pi', v)$ such that:

$$\begin{array}{l} C \ = \ \pi, v \vdash s_0 \longrightarrow s_1 \longrightarrow \ldots \longrightarrow s_t \ , \\ C' \ = \ \pi', v \vdash s'_0 \longrightarrow s'_1 \longrightarrow \ldots \longrightarrow s'_r \end{array}$$

and

$\forall 0 \leq i \leq t : s_i = (p_i, \sigma_i)$ **and**
$\forall 0 \leq i \leq r : s'_i = (p_i, \sigma'_i)$.

Define $C \mathcal{R} C'$ if and only if $t = r$ and for any $0 \leq i \leq t$:

1. $p_i = p'_i$
2. $\forall j < i : p_j \neq p \Rightarrow \sigma_i = \sigma'_i$ **and**
3. $\exists j < i : p_j = p \wedge \sigma_i \backslash \mathrm{x} = \sigma'_i \backslash \mathrm{x}$

The bisimulation conditions are easily verified.

CONSTANT FOLDING

$$\mathbf{x{:=}y} \quad \Longrightarrow \quad \mathbf{x{:=}c}$$

if

$$\overleftarrow{A}\,(\ \neg\mathtt{def}(\mathbf{y}) \wedge \neg\mathtt{stmt}(\mathbf{read\ x})\ U\ \mathit{stmt}(\mathbf{y{:=}c}) \wedge \mathit{conlit}(\mathbf{c}))$$

In this case relation $\mathcal{R}$ is the identity relation(!)

That is, we wish to prove that for any length n, a computation prefix of π of length n is equal to a computation prefix of π' with the same length.

Statement and proof of conditions: **slightly more complex.**

CODE MOTION/LOOP INVARIANT HOISTING

p : skip $\Longrightarrow$ x:=e

q : x:=e $\Longrightarrow$ skip

if

$p \models A(\neg use(\mathsf{x})\ W\ node(q))$

$q \models \neg use(\mathsf{x}) \wedge$

$\overleftarrow{A}((\neg def(\mathsf{x}) \vee node(q)) \wedge$

$trans(\mathsf{e}) \wedge \neg \mathtt{stmt}(\textbf{read x})\ W\ node(p))$

Suppose

$$C = \pi, v \vdash (p_0, \sigma_1) \rightarrow \ldots \rightarrow (p_t, \sigma_t)$$
$$C' = \pi', v \vdash (p'_0, \sigma'_1) \rightarrow \ldots \rightarrow (p'_{t'}, \sigma'_{t'})$$

Define the $\mathcal{R}$ relation as: $C\mathcal{R}C'$ if and only if $t = t'$, $p_i = p'_i$ for all $0 \leq i \leq t$ and one of the following cases holds:

1. $\sigma_t = \sigma'_t \ \wedge \ \forall 0 \leq i < t : p_i \notin \{p, q\}$
2. $\sigma_t = \sigma'_t \ \wedge \ \exists 0 \leq i < t : p_i = q \ \wedge \ \sigma_i = \sigma'_i \wedge \forall i < j < t : p_j \notin \{p, q\}$
3. $\exists 0 \leq i < t : p_i = p \ \wedge \ (\sigma_t \setminus \mathbf{x} = \sigma'_t \setminus \mathbf{x}) \wedge (\sigma_i \setminus \mathbf{x} = \sigma'_i \setminus \mathbf{x}) \ \wedge \ \forall i < j < t : p_j \notin \{p, q\}$

EXCERPT FROM THE END OF THE PROOF

(No, you're not morally obliged to read and understand the details!) Suppose

$$C = \pi, v \vdash (p_0, \sigma_1) \longrightarrow \ldots \longrightarrow (p_t, \sigma_t)$$
$$C' = \pi', v \vdash (p'_0, \sigma'_1) \longrightarrow \ldots \longrightarrow (p'_{t'}, \sigma'_{t'})$$

Assumptions:

	π		π'
$p_i = p$	`skip`	$\Rightarrow$	`x:=e`
$p_t = q$	`x:=e`	$\Rightarrow$	`skip`

and $C \mathcal{R}_3 C'$. Then x maps to the same value in σ_{t+1} and σ'_{t+1}:

$$\begin{aligned}
\sigma_{t+1}(\mathsf{x}) &= [\![\mathsf{e}]\!]\sigma_t && \text{(semantics of } \mathcal{I}_{p_t} = (\mathrm{x}{:=}\mathrm{e})) \\
&= [\![\mathsf{e}]\!]\sigma_{i+1} && \text{(argument in paper)} \\
&= [\![\mathsf{e}]\!]\sigma_i && \text{(semantics of } \mathcal{I}_{p_i} = \mathtt{skip}) \\
&= [\![\mathsf{e}]\!](\sigma_i \setminus \mathsf{x}) && \text{(since } \mathrm{x} \notin \mathit{vars}(\mathrm{e})) \\
&= [\![\mathsf{e}]\!](\sigma'_i \setminus \mathsf{x}) && \text{(since } C \mathcal{R}_3 C' \Rightarrow \sigma_i \setminus \mathrm{x} = \sigma'_i \setminus \mathrm{x}) \\
&= [\![\mathsf{e}]\!](\sigma'_i) && \text{(since } \mathrm{x} \notin \mathit{vars}(\mathrm{e})) \\
&= \sigma'_{i+1}(\mathsf{x}) && \text{(semantics of } \mathcal{I}'_{p_i} = (\mathrm{x}{:=}\mathrm{e})) \\
&= \sigma'_t(\mathsf{x}) && \text{(argument in paper)} \\
&= \sigma'_{t+1}(\mathsf{x}) && \text{(semantics of } \mathcal{I}'_{p_t} = \mathtt{skip})
\end{aligned}$$

Thus $\sigma_{t+1} = \sigma'_{t+1}$.

SUMMARY

- We specified transformations as rewrite rules with temporal logic formulas as side conditions.
- We set up a framework in which temporal logic plays a central role in correctness proofs of classical compiler optimizing transformations.
- Describing transformations as rewrite rules is not new; but use of temporal logic to specify when to apply them, and to prove their validity, seems to be new.

To prove correctness: show that a transformation does not change semantics, i.e., $[\![\pi]\!] = [\![\pi']\!]$.

The only creative part is finding the relation $\mathcal{R}$.

This relation is closely related to the temporal logic side conditions of the transformation. The remainder of the proof is usually straightforward.

IMPERATIVE PROGR. TRANSF. BY REWRITING

Part of a larger project (David Lacey, Eric Van Wyk, Oege de Moor) to study

- declarative methods of specifying optimizations and
- means of automatically generating optimizers from these specifications.

Specifications of transformations are rewrite rules with temporal logic side conditions, implemented by a graph rewriting system and model checker.

Very simple programming language so far.

- Adding new statement types does not affect the effectiveness of our method.
- Exceptions and procedures would, however, require changes to the control flow model and the transition systems used in the proof.
- The specification of the transformations however does not dramatically change.

FUTURE WORK

- Systematic derivation **of the bisimulation relation** $\mathcal{R}$ **from rewrite rule:** $\mathcal{I} \Rightarrow \mathcal{I}'$ if ϕ
- **Is it** decidable **whether rewrite rule** $\mathcal{I} \Rightarrow \mathcal{I}'$ if ϕ **is valid, i.e., preserves semantics of all programs ?**
- **Is it possible to derive the weakest side condition, given a rewrite rule?**
- **What is the** efficiency **of program transformation by such rule-based rewriting ? Can it be engineered to be of** comparable efficiency **with traditional compiler optimizations?**
- **Improved specification language.**
- **Conjecture: This can be applied to supercompilation (cf. Geoff, Gavin)**

FUTURE WORK II

Our language: rather like a traditional compiler's "intermediate language".

The method can validate many traditional optimizing compiler transformations, e.g. in

- the "Dragon book" or Muchnick's compendium.

Lerner, Millstein and Chambers saw the January 2002 POPL paper. By November 2002 they had designed

- a stronger compiler intermediate language with pointers and procedures
- a language "Cobalt" to specify optimization rewrite rules (a domain-specific language, weaker than the CTL that we use)
- Implemented their strategy with Simplify, an automatic theorem prover
- Found lots of subtle bugs in enabling conditions

Since then: Rhodium, inferring optimiser flow functions from semantics.

RELATED WORK

- **J. Knoop, O. Rüthing, B. Steffen.** Optimal Code Motion: Theory and Practice. **ACM TOPLAS, 16(4):1117–1155, 1994.**
- **D.A. Schmidt, B. Steffen.** Program analysis as model checking of abstract interpretations. **Proc. 5th SAS, 1998.**
- **D. Kozen and M. Patron.** Certification of compiler optimizations using Kleene algebra with tests. **In CL2000, LNAI vol. 1861, Springer-Verlag,, 2000, pp. 568–582.**
- **D. Lacey and O. de Moor.** Imperative program transformation by rewriting. **In Compiler Construction, vol. 1113 of LNCS, pp. 52–68. Springer-Verlag, 2001.**
- **S. Lerner, T. Millstein, C. Chambers.** Automatically proving the correctness of compiler optimizations. **ACM PLDI, pp. 220-231, 2003.**
- **Lerner et al** Automatic Inference of Optimizer Flow Functions from Semantic Meanings. **PLDI, 2007.**